2009 33rd Annual IEEE Software Engineering Workshop

Towards an Alloy Formal Model for Flexible Advanced Transactional Model Development

Barbara Gallina, Nicolas Guelfi, Pierre Kelsen
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi,
L-1359 Luxembourg
{barbara.gallina, nicolas.guelfi, pierre.kelsen}@uni.lu

Abstract

SPLACID is a semi-formal language conceived for the specification and synthesis of (advanced) transactional models from basic features, such as transaction types and (relaxed) ACID variants. SPLACID is an improvement of the ACTA framework offering a well-structured and formal syntax. Neither ACTA nor SPLACID, however, benefit from a formal tool-supported semantics. This paper presents the first step for having a full formal semantics of SPLACID by translation to Alloy. In particular, we present the translation of the SPLACID concepts into Alloy concepts focusing on those concepts pertaining to the structure of a Transactional Model and those characterizing the isolation variant. The Alloy specification obtained by this translation preserve the SPLACID main key-properties, namely, modularity, flexibility and reusability. To support this claim we show how flexible, modular and reusable structures and isolation variants can be obtained in Alloy. Finally, we analyze the flat and nested transactional model structures and the serializability-based isolation variant using the Alloy Analyzer.

Keywords
ACID properties, Relaxed ACID properties, dependability, reusability, Software Product Line, ACTA framework, formal semantics, Alloy language.

1. Introduction

The quality of a distributed concurrent computation and, in particular, its reliability may be increased through transactional principles [11]. Since transactional principles establish constraints which may reduce dramatically the number of allowed computations, to be applicable in nowadays distributed systems, they have to offer a certain degree of flexibility [7]. The ACID (Atomicity, Consistency, Isolation and Durability)-based Transactional Model (TM), called Flat transactions, has been recognized to be too rigid and limited from a functionality point of view when used for application domains having requirements in contrast with those typically expected for transactions [9]. Several advanced TMs (i.e. Nested transactions) have been introduced to overcome functionality-related limitations [6]. These models differ from each other in the way ACID properties are relaxed. The SPLACID language [8] has been introduced to offer a means to specify TMs in a flexible and Software Product Line (SPL)-oriented way. SPLACID is a semi-formal language conceived for the specification and synthesis of TMs from basic features, such as transaction types and (relaxed) ACID variants. SPLACID, therefore, allows users to structure in a modular, reusable and flexible manner the constraints which are necessary to limit the allowed computations. SPLACID targets two groups of users: transactional engine developers who might use SPLACID to specify new advanced transactional models; transactional application developers who might use SPLACID to specify the adequate TM with respect to the application needs.

SPLACID is an improvement of the ACTA framework [4]. The ACTA framework represented the first effort towards the provision of a means for the specification and synthesis of TMs. The ACTA specifications, which consist of a list of semi-formal axioms, do not present a clear structure. SPLACID improves the ACTA framework by offering a well-structured and formal syntax. Neither ACTA nor SPLACID, however, benefit from a complete formal semantics. To achieve a formal semantics for SPLACID and more specifically an executable (tool-supported) semantics, we translate SPLACID specifications into Alloy [13] specifications and exploit the tool support, provided by the Alloy Analyzer, to carry on analysis.

For our purposes, Alloy is the right candidate for the following two main reasons: 1) it is a light-weight formal language, suitable at the early stages of the software development to identify the right software abstractions; 2) it is equipped with a powerful logic which includes the transitive closure operator. In our context, having a means to identify the right software abstractions is fundamental because it makes easier: 1) the development of a transactional engine supporting an advanced TM; 2) the selection of the adequate TM with respect to the application needs in case of transactional application development. The transitive closure operator is also fundamental because often analyz-

ing abstractions related to transactional principles means satisfying specific reachability constraints. For instance, to reason about concurrency control abstractions, the transitive closure operator is used to express the acyclicity constraints in the graph representing the dependencies among transactions. Acyclicity ensures serializability (see i.e. [1]).
In this paper, we present the first step aimed at achieving a full SPLACID semantics in Alloy. In particular, we focus on the translation of those SPLACID concepts pertaining to the structure of a TM and on those characterizing the isolation variant.
The rest of the paper is organized as follows. Section 2 introduces the background information. Section 3 proposes an Alloy semantics for SPLACID. Section 4 explains how to achieve modular, flexible and reusable specifications for TMs in Alloy. Section 5 uses the Alloy Analyzer to find instances satisfying the constraints which have to hold at the boundary of the Flat and Nested transactions model and also constraints which have to hold to guarantee serializability. Section 6 discusses related work. Finally, Section 7 presents some concluding remarks and future work.

2. Background

This section briefly introduces the background on which we build our contribution. The section is organized as follows. Sub-section 2.1 illustrates the commonalities and the variabilities of the Software Product Line (SPL) constituted of Transactional Models (TMs), called TMsPL. Sub-section 2.2, presents the SPLACID language and its language constructs suitable to specify the abstractions, which represent the commonalities and the variabilities of the TMsSPL. Finally, Sub-section 2.3, introduces the Alloy formal language and its interesting features which make it appealing for providing an executable semantics.

2.1 Commonalities and variabilities in TMsPL

TMs, as discussed in [7], may be better understood and compared by adopting an SPL perspective. SPL glasses enable the identification of commonalities and variabilities among TMs. Before revealing the commonalities and variabilities among TMs, in what follows, we recall the basic SPL terminology.
A *commonality* is the ability of an asset to be maintained as a constant; while a *variability* is the ability of an asset to be changed (customized) for use in a particular context. Variabilities are defined through variation points and variants. A *variation point* is the place within an artefact where a decision can be made [10]. *Variants* represent the alternatives associated to a variation point. A *Feature Diagram* (FD) is a graphical representation of a hierarchically arranged set of features (properties of a system). Features might be mandatory (in case of commonalities), optional or alternative (in case of variabilities). A cardinality-based FD is an FD annotated with cardinalities [5].
Advanced TMs often introduce a structure in the “flatland” represented by the initial TM, as well as a different “ACID-ity” (that is different notions of ACID properties, ACID variants, which might be combined). A TM, as the partial cardinality-based FD in Figure 1 summarizes, is commonly structured as collection of inter-dependent transaction types. Each transaction type is characterized by commonalities (constraints constraining the transaction boundary, coexistence of an Atomicity variant, a Consistency variant, an Isolation variant and a Durability variant) and variabilities represented by the ACID variation points. At the Isolation variation point, for instance, a pre-defined Isolation variant [1] might be selected or a new variant might be composed by selecting a view type, a conflict type, etc. Similarly, at the other variation points, pre-defined variants might be selected or new variants might be introduced by selecting and combining variants pertaining to the ACD variants sub-features. The complete introduction of all the features which appear in the cardinality-based FD, shown in Figure 1, is, however, outside of the scope of this paper. The interested reader may infer them by reading [8].

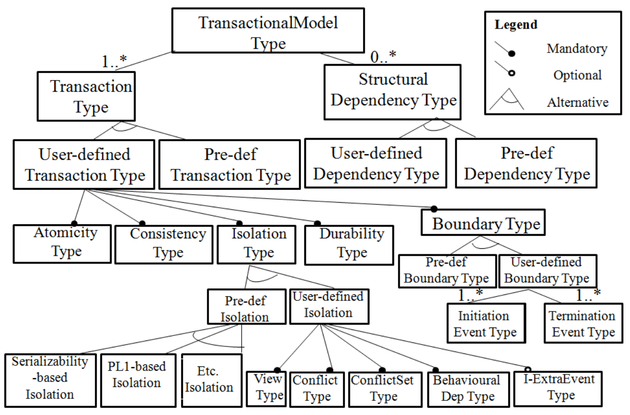


Figure 1 Cardinality-based FD of the TMsPL

2.2 SPLACID

SPLACID [8] is a semi-formal language designed to specify and synthesize TMs on the basis of their fundamental properties. SPLACID has been conceived to be used within the (under construction) SPL-oriented requirements engineering process, called PRISMA, to support the specification activity. SPLACID is based on ACTA [4], a unified framework conceived for the specification, synthesis and verification of TMs from building blocks. The main advantage of SPLACID over ACTA is that it offers a well-structured concrete syntax, given in BNF like form, which integrates the SPL perspective extensively described in [7] and briefly presented in the previous sub-section. SPLACID makes it possible to structure a specification of a TM by selecting, whenever possible, reusable features. In the following, we recall some syntactical SPLACID rules.

*1- < **TransactionalModelType** >::=< TMTId >*
< TransactionType >$^{+}$ *< StructuralDependencyType >**

The above recalled rule defines a TM. A TM is defined as a sequence of: 1) the TM identifier, 2) one or several transaction types; 3) zero or more structural dependency types. A TM represents a directed multi-graph having as nodes a set of transaction types and as directed edges a set of structural dependency types, constraining the structure.

2- $< \textbf{\textit{StructuralDependencyType}} > ::= < Predef\_SDepID >|$
$< TransTypeId >< SDepID >< TransTypeId > \Leftrightarrow < Predicate >$

The above recalled rule defines a Structural Dependency Type which relates two transaction types (forming the edge) in case a predicate is satisfied. As a result, a TM may represent simple TMs such as *Flat transactions* (which have no structure and therefore are represented by a graph having only one node, the *Flat* type, and an empty set of edges), but also complex advanced TMs. For instance, the *Nested transactions* TM has two transaction types (*Root* and *Child*) inter-related through four dependencies: two to establish initiation dependency and the other two to establish termination dependency (i.e., child initiation follows the root initiation; the root termination follows the child termination). The combination of initiation and termination dependencies is equivalent to a composed structural dependency identifying containment.

One composed structural dependency could therefore connect the root type with the child type specifying that every transaction of type child must be contained within its root. Another composed structural dependency could connect the child type to itself specifying that every transaction of type child invoked by another transaction of type child has to be contained in it.

3- $< \textbf{\textit{TransactionType}} > ::= < Predef\_TransTypeId >|$
$< TransTypeId >:< BoundaryType >< AVariantType >$
$< CVariantType >< IVariantType >< DVariantType >$

The above-recalled rule defines a transaction type. A transaction type is defined either as a pre-defined transaction type identifier or as a sequence of: 1) a transaction type identifier, 2) the boundary type (identifying the types of events that mark the initiation and termination of each transaction of that type); 3) the ACID variants which represent the properties in terms of ACIDity of the transaction type.

4- $< \textbf{\textit{BoundaryType}} > ::= < Predef\_BoundaryId >|$
$< BoundaryId >:< IEset >< TEset >< pre-post-Boundaries >$

The above-recalled rule, defines the boundary of a transaction type. The boundary is defined either as a pre-defined boundary type identifier or as a sequence of: 1) the boundary identifier; 2) the set of Initiation Event types; 3) the set of Termination Events types; 4) the pre and post conditions of the event types.

In SPLACID, the specifications of the ACID properties are nicely modularized. As a result, it is possible to reason about each property in isolation, and even to write specifications that describe different variants of a property. Once a "library" of property specifications is available, once the SPL domain is engineered, a TM can be built in SPLACID simply by assigning the desired ACID property variants to the transaction types it is composed of.

In the following, we recall the rule defining the isolation variant.

5- $< \textbf{\textit{IVariantType}} > ::= < Predef\_IVariantId >|< IVariantId >:$
$< ViewType >< ConflictMatrixType >< ConflictSetType >$
$< BDType >^{*} < I\_TE-pre-post >^{+} [< I-XEset >]$

The isolation variant is defined either as a pre-defined variant identifier (in case of reusable, well-known [1], isolation variants) or as the sequence of: 1) the variant identifier; 2) the view type (which defines the visibility associated to a transaction); 3) the conflict matrix type (which defines a compatibility matrix between operation types accessing the same object); 4) the conflict set type (which defines the set of object events to which a transaction has to pay attention); 5) the eventual behavioural dependencies (which define the ordering constraints due to the occurrence of the events accessing objects); 6) the pre and post conditions related to isolation characterizing the Termination Events; 7) the eventual extra significant events, which influence the interference.

The complete introduction of the rules defining all the non-terminals presented in the above-introduced rules is outside of the scope of this paper. The interested reader may refer to [8]. In the following, we present the ideal specification of Flat transactions in SPLACID in which reusability is maximized:

Flat transactions-
Flat /*one transaction type only composes the TM*/
Where Flat is further specified as:
StandardBoundary /*reuse of a pre-defined boundary type*/
FailureAtomicity /*reuse of a pre-defined atomicity type*/
FullConsistency /*reuse of a pre-defined consistency type*/
SerializabilityBasedIsolation /*reuse of a pre-defined isolation type*/
StrictDurability /*reuse of a pre-defined durability type*/

In SPLACID, reuse might be maximized by selecting and composing reusable pieces of specifications (identified by pre-defined identifiers). These pieces are the result of a domain engineering phase following SPL engineering practices (as prescribed within the, under construction, PRISMA process).

The semantics of a SPLACID specification coincides with the History, the strict partial order of events modelling the computation. The semantics of the different constructs of the SPLACID language consists of a set of constraints on the ordering relation existing among events and on the events themselves. In the above introduced SPLACID specification, for instance, the StandardBoundary imposes that: 1) the set of Initiation Event types is composed of a single element, called InitiateType; 2) the set of Termination Event types is composed of a single element, called TerminateType; 3) each transaction having a transaction type characterized by the StandardBoundary must satisfy a standard constraint. This constraint requires that: transactions must have exactly one event of type Initiation Event Type; exactly one event of type Termination Event Type; the event of type Initiation Event Type has to be the first one; the one of type Termination Event Type has to be the last one; events which do not mark the transaction boundary have to follow the event of type Initiation Event Type and precede the event of type Termination Event Type.

2.3 Alloy

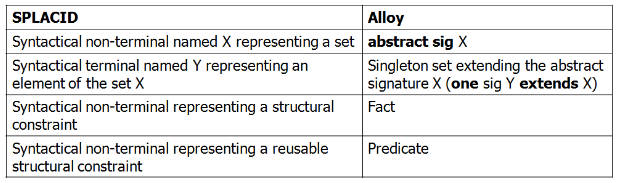
Alloy [13] is a formal language suitable at the early stages of software development to identify the right software abstractions. Alloy's logic is a relational logic which combines the quantifiers of first order logic with the operators of the relational calculus (i.e. the transitive closure operator, fundamental for reachability properties analysis). Alloy's logic permits to express complex structural and behavioural constraints. Alloy's logic is undecidable. However a tool, called Alloy Analyzer, to analyze the soundness of the Alloy models is available. This tool supports the analysis under a given scope (its analysis is complete up to the scope) and it looks for some assignment to variables that makes the constraints true. Once an assignment is found, instances (models) satisfying the constraints can be visually shown. If an assignment is not found the only conclusion is that the constraints cannot be solved within the chosen scope. The scope defines a multidimensional space of test cases, each dimension corresponding to the bound on a particular type (maximum number of instances allowed for a particular type). The scope specification is separated by the model and this separation permits to adjust the scope in a fine-grain manner without modifying the model. Despite the incompleteness of the Alloy analysis, the "small scope hypothesis" (according to which, the analysis of all small cases increases the chance of covering meaningful test-cases) encourages the fundamental but delicate work of sizing the scope.
An Alloy model (specification) is a structured specification made of signatures, relations, facts, predicates, functions, assertions and commands. A *signature* introduces a basic type (a typed set of atoms). Signatures have an optional body constituted of a collection of relations called *fields*, each with a fixed type. Signature extension is a powerful feature to support classification hierarchy. *Facts* are explicit constraints on relations and signatures. Facts always hold. They have therefore to be verified by all the instances of the model. *Predicates* define reusable constraints. *Functions* define reusable expressions. *Assertions* introduce a constraint that is intended to follow from the facts of the model. Commands (*check* and *run*) give directions to the Alloy Analyzer tool during its analysis and they allow users to carry on two distinct kinds of analysis: simulation (running of a predicate) and assertions checking. These two kinds of analysis reduce to the same analysis problem discussed above concerning assignment finding.

3. An Alloy semantics for SPLACID

So far, the SPLACID language offers only a well-structured concrete syntax; no formal semantics is yet available. In this section, we provide a first step towards a complete formal semantics for SPLACID. To obtain the semantics, we translate SPLACID specifications into Alloy specifications and then we exploit the Alloy Analyzer tool to carry on satisfiability analysis.

Mapping SPLACID concepts into Alloy

To achieve an Alloy specification from a SPLACID specification, a translation has to be carried out. In the following we give some general translation guidelines.
For each syntactical non-terminal representing a set (i.e. TransactionalModelType, TransactionType, StructuralDependencyType, etc.), an abstract signature is created in Alloy. This translation is motivated by the fact that a non-terminal represents an abstract construct used to categorize terminals and no instance is needed for it. In Alloy, an abstract signature has no elements except those belonging to its extensions. Therefore an abstract signature adequately represents a non-terminal.
For each syntactical terminal representing an element of a set (i.e. Flat is a concrete Transaction Type), an enumeration having only one element is created in Alloy (one Sig, a singleton set).
A sequence is mapped into a set of fields (fields are relations in Alloy). In case of a sequence of non-terminals, for each non-terminal (except those representing identifiers or constraints) a field is created which relates it with the abstract signature representing the non-terminal on the left-side of the rule. Each multiplicity indication is mapped in the equivalent multiplicity indication in Alloy (i.e. '+' is mapped into *some*).



SPLACID	**Alloy**
Syntactical non-terminal named X representing a set	**abstract sig** X
Syntactical terminal named Y representing an element of the set X	Singleton set extending the abstract signature X (**one** sig Y **extends** X)
Syntactical non-terminal representing a structural constraint	Fact
Syntactical non-terminal representing a reusable structural constraint	Predicate

Figure 2 Mapping SPLACID concepts into Alloy concepts
Figure 2 partially presents the mapping that we have identified between SPLACID concepts and Alloy concepts.
In the following we apply the mapping rules limiting our attention to the concepts useful for the definition of the structure of a TM and to those useful for the definition of the isolation variant. We then introduce informally some of the constraints which constrain the SPLACID concepts and we explain how to map them into Alloy facts or predicates. The Alloy code presented in the paper is directly commented in terms of Alloy comments ('--').

Focus on structure-related concepts

Following the first mapping rule shown in Figure 2, the non-terminal called TransactionalModelType, representing the set of TMs, is mapped into the abstract signature called TransactionalModelType. This abstract signature has two fields which relate the abstract signature itself to the two further abstract signatures TransactionType and StructuralDependencyType, corresponding to the two non-terminals written in the right-side of the syntactical rule 1 (defined in Sub-section 2.2).

```
abstract sig TransactionalModelType {
  --an abstract signature has no elements
  --except those belonging to its extensions
  t: some TransactionType,
  --non empty set which contains the nodes of
  --the graph defining the TM structure
  dependencies: set StructuralDependencyType
  --set which contains the edges of the graph
}
```

In the following we present the Alloy code obtained by applying the first mapping rule to the other structure-related concepts.

```
abstract sig TransactionType {
  boundaries: BoundaryType,
  iVariant:IVariantType,
  --dVariant: DVariantType, aVariant: AVariantType,
  --cVariant: CVariantType, still in progress
}
abstract sig StructuralDependencyType{
  sd: TransactionType->TransactionType
}
abstract sig BoundaryType {
  iEventTypes:set InitiationEventType,
  --set of event types to mark a transaction initiation
  tEventTypes:set TerminationEventType
  --set of event types to mark a transaction termination
}
```

The TransactionalModelType extensions may generate flexible TMs according to the needs, by varying, for instance, the Transaction Types which compose it and their ACIDity. Potentially all the TMs which differ on the basis of the structure and ACIDity might be specified.

Focus on Isolation-related concepts

In the following, we present the Alloy code defining the abstract signatures related to isolation. Following the first mapping rule shown in Figure 2, we obtain the following translation:

```
abstract sig IVariantType {
  view: ViewType,
  conflict: ConflictMatrixType,
  conflictSet: ConflictSetType,
  bd: BDType,
  iXETypes : set IExtraEventType,
}
```

The IVariantType extensions may generate a spectrum of isolation degrees according to the needs, by varying the ConflictMatrixType or the BDType, etc. (see Section 2.2, rule IVariantType). This guarantees flexibility and reusability of specific sub-features.

```
pred IVariantType::relates[t1,t2: Transaction] {
--This predicate composes the IVariant sub-features and is
--used to define the dependence graph
        t1->t2 in this.bd.d some e1: t1.eventSet &
this.view.viewEvents[t2] & this.conflictSet.conflictEvents[t2]|
some e2: t2.eventSet & this.view.viewEvents[t1] &
this.conflictSet.conflictEvents[t1]|
                e1.op.type->e2.op.type in this.conflict.m
}
pred IVariantType::acyclic {
--This predicate imposes acyclicity in the dependence graph
        let r= { t1: Transaction, t2: Transaction |
        this.relates[t1,t2]}| no ^r & iden
}
```

```
--defines the visibility of a transaction
abstract sig ViewType{
   v: Transaction -> one EventOrder,
 --associaciates to each transaction a strict partial order of events
}
--defines a relation between operation types
abstract sig ConflictMatrixType{
  m: OperationType-> OperationType,
}
abstract sig BDType{
  d: Transaction -> Transaction,
--establish dependencies between transactions due to behaviour
}
abstract sig ConflictSetType{
  c: Transaction-> Event,
  --associates Object Events to the transaction
} { (Transaction.c).type=ObjectEventType}
```

Mapping SPLACID constraints into Alloy

The set of allowed SPLACID specifications is obtained by mapping also syntactical non-terminals representing structural constraints. In the following, we present the predicate which translates the standard constraint (see Section 2.2) which has to hold at the boundary:

```
pred BoundaryType::standardAxiomsHold[eposet: EventOrder] {--eposet is a strict partial order of events
one e1: eposet.domain | one e2: eposet.domain|all
e3:(eposet.domain -(e1+e2)) | (e1&e2)=none and
-- one initiation event and first event
(e1.type in this.iEventTypes) and (eposet.eventSource[e1])
-- one termination event and last event
and (e2.type in this.tEventTypes) and (eposet.eventSink[e2])
and      eposet.precedes[e1,e2]
--all object events have to follow the event of type initiation
-- event and precede the one of type termination event
and (e3.type=ObjectEventType) and eposet.precedes[e1,e3]
and eposet.precedes[e3,e2]
}
```

In the following we present the predicate which translate the constraint which has to hold to have a structural dependency:

```
pred StructuralDependencyType::relatesStructurally[tt1,tt2:
TransactionType] {
--a structural dependency implies the existence of an
--ordered pair of events (initiation or termination) belonging
--to two different transactions
(tt1->tt2) in this.sd=>
(some e1:Event|some e2:Event|some t1: Transaction| some
t2:Transaction| no (e1 &e2) and  no (t1 &t2) and e1.type in (
tt1.boundaries.iEventTypes + tt1.boundaries.tEventTypes )
and e2.type in ( tt2.boundaries.iEventTypes +
tt2.boundaries.tEventTypes) and t1.type=tt1 and t2.type=tt2
and e1 in t1.eventSet and e2 in t2.eventSet=>
(e2->e1 in History.r))
}
```

Both predicates are completely reusable since they are defined at the abstract (meta-model) level. The interested reader may refer to the report [17] for further examples.

4. Specifications of TMs in Alloy

The Alloy abstract signatures, representing the non-terminals of the SPLACID grammar (the abstract meta-classes of the SPLACID meta-model), are extended in this section to define modelling concepts which often represent pre-defined and reusable features (concrete meta-classes). These features allow users to define TMs in a modular, flexible and reusable way, according to the specific application domain needs.

In what follows, we illustrate the Alloy code obtained by following the second mapping rule shown in Figure 2. In particular, we illustrate the translation of the pre-defined modelling concepts related to the Flat and Nested transactions models and to their structure. In so doing, we see that the pre-defined boundary type, called StandardBoundaryType, is reused to characterize all the transaction types composing the two models.

Structure: Flat transactions

Flat transactions represent an element of the set of possible TMs. Following the second mapping rule shown in Figure 2, we translate this element in Alloy as a singleton set.

```
one sig FlatModel extends TransactionalModelType {}
--the sig FlatModel contains a single atom
{ --represents a fact signature
   t = Flat and no dependencies,
   --that is a constraint which always has to hold
   --requiring the multi-graph representing the FlatModel
   --to be a single node (the Flat TransactionType)
}
```

In the following, we present the Alloy code of the signatures which define the Flat transaction type.

```
one sig Flat extends TransactionType{} {
  boundaries= StandardBoundaryType --reuse enforcement
  --iVariant will be later discussed and analyzed
}
one sig StandardBoundaryType extends BoundaryType{} {
  --the set of InitiationEvent types has to equal the InitiateType
  iEventTypes = InitiateType
  -- the set of tEvent types has to equal the TerminateType
  tEventTypes = TerminateType
all t1:Transaction| t1.type.boundaries = this =>
(boundaries.standardAxiomsHold[t1.eventOrder])
--reuse of the standard constraint introduced in Section 3
and #t1.eventOrder.domain >=2
}
```

Structure: Nested transactions

Nested transactions also represent a concrete element of the set of possible TMs. Therefore in Alloy a singleton set is used. In the following, we present the Alloy code of the NestedModel signature. Root and Child represent the two transaction types which compose the nested model. Four edges relate the two transaction types and these edges represent structural dependencies establishing containment relationship (as explained in sub-section 2.2).

```
one sig NestedModel extends TransactionalModelType {}{
t = Child + Root
dependencies= {ChildBeginDependsOnRoot + RootTermina-
tionDependsOnChild + ChildBeginDependsOnChild + Child-
TerminationDependsOnChild
}
fact{all e:Event |e in History.domain =>some t:Transaction |
t.type in Root+Child and e in t.eventSet}
--a child has to have a parent!
fact {all e:Event|some t: Transaction |(e in History.domain
and e in t.eventSet and t.type=Child)=>
some t1:Transaction|t1.type = Root+Child and e.source=t1}
}
one sig Child extends TransactionType{}{
boundaries= StandardBoundaryType --reuse enforcement
--each transaction of type child has a parent
--their events are invoked by a source
all e: Event|all t:Transaction|(e in t.eventSet  and t.type=this)
=>(one t1:Transaction| no (t1&t) and t1.type in Root +Child
and t1= e.source)
}
one sig Root extends TransactionType {}
{
boundaries= StandardBoundaryType --reuse enforcement
all t: Transaction| t.type = this =>(all e:Event| e in t.eventSet
=> no e.source)
}
```

For brevity we do not present all four dependencies but only one, the other three are similarly defined.

```
one sig ChildBeginDependsOnChild extends BeginDep{}{
sd=Child->Child
relatesStructurally[Child,Child]
(all t1: Transaction | all t2: Transaction| (t1.type=Child and
t2.type=Child and no (t1 &t2) and t1.iEventSet.source = t2) =>
(all e1: t1.iEventSet | all e2: t2.iEventSet |(e2->e1) in Histo-
ry.r))
}
```

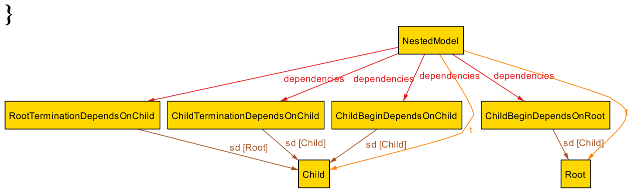


Figure 3 Multi-graph representing Nested Transactions

Isolation Variant (towards flexible ACIDity)

The abstract signatures, useful for the definition of the Isolation Variant, are here extended to define a specific type of Isolation namely traditional serializability-based Isolation. In the following we present the Alloy code of the signatures needed to define serializability-based Isolation.

```
one sig SerializabilityBasedIsolation extends IVariantType{}{
            view = FullView
            conflict = SyntaxBasedConflictMatrix
            conflictSet = StandardConflictSetType
            bd = allBD
           no iXETypes
}
one sig FullView extends ViewType{} {
all t:Transaction | one eo: EventOrder | eo.equals[History]
and t->eo in v
}
--two operations conflict if at least one is a write
one sig SyntaxBasedConflictMatrix extends ConflictMatrix-
Type{}{
           m= Write->Write + Write-> Read + Read->Write
}
--a dependency between two transactions exist when there are
```

```
--two events of type ObjectEventType (each of which belonging
--to a single transaction) which both trigger an operation on the
--same object and are related through the ordering relation.
```
one sig allBD extends BDType{} {
d={t1:Transaction, t2:Transaction| some disj e1,e2:Event| (e1.op.ob = e2.op.ob and History.precedes[e2,e1] and e1 in t1.eventSet and e2 in t2.eventSet and e1.type=ObjectEventType and e2.type=ObjectEventType)}
}
```
--to each transaction are associated those events of type
--ObjectEventType which do not belong to the transaction
--domain
```
one sig StandardConflictSetType extends ConflictSetType{} {
c= {t: Transaction, e: Event | e.type=ObjectEventType and not e in t.eventSet }
}

In traditional serializability, each transaction has a full view. In fact, in-place update is always implicitly assumed (see our FullView signature). Conflicts between operations (read-write) are identified on the basis of a syntactic-based compatibility matrix. If at least one operation is a write, a conflict is identified. To protect itself from interference, each transaction has to pay attention to all the object events invoked by all other transactions (conflict set definition).

The above provided singleton sets related to isolation (which represent reusable features) might be reused to obtain other isolation variants. To obtain the PL1 [1] based Isolation variant, for instance, all the singleton sets characterizing the serializability-based Isolation variant might be reused, except for the *allBD* BDType which has to be replaced by a different dependency type (see WriteWrite dependency in [1]).

5. Satisfiability analysis

The Alloy Analyzer tool may be used to analyze the model automatically. Two types of analysis can be carried out: simulation and checking. In this section, we only discuss simulation analysis to show the satisfiability of the predicates of our Alloy model. In particular, we ask the tool to show consistent instances satisfying very simple but important predicates. In the first two analyses we do not take into consideration the isolation variant (the *iVariant* field is commented). We only look at constraints related to the boundary. In the third analysis we focus on a specific isolation variant, namely *serializability-based Isolation*.

Before presenting the analysis which has been carried out, we introduce further elements of the Alloy model, needed to the analysis purposes.

Once a TM has been specified, a transactional computation is the combination of the business needs and the desired TM. A transactional computation exhibits events belonging to the business needs (events of type *ObjectEvent*, which, in a first phase, can be limited to events executing read and write operations) and events belonging to the TM management, known as events of type *SignificantEvent*. Figure 4 illustrates the taxonomy related to the event types.

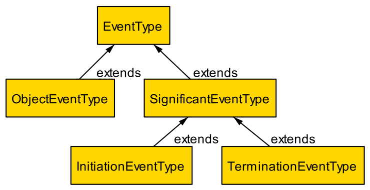


Figure 4 Meta-model representing the event types hierarchy

In the following, we present the Alloy code useful to specify events:

sig Event {
type: EventType,
op: lone Operation,
```
--to each Event is associated at most 1 operation
```
source: lone Transaction,--represents the invoker
} {--an operation is associated only in case of events of type
```
--ObjectEvent
```
some t: Transaction| this in t.eventSet type in (InitiationEventType+TerminationEventType)=>no op
type in ObjectEventType=>one op
```
--a transaction is not allowed to be the invoker/parent of its own
--events
```
some t: Transaction| this in t.eventSet=>not (source=t)
}

In the following, we present the Alloy code used to specify the business needs:

sig BusinessEvents {
```
--represents the business events, the set of read/write operations
--to be executed to achieve the business goal
```
domBE: set Event,
}{
all e: Event | e in domBE=> e.type = ObjectEventType
some t: Transaction| this=t.bE and domBE in t.eventSet
}

fact businessEventsHaveDisjointEvents {
```
--the intersection between the domains has to be empty
```
all disj be1, be2: BusinessEvents | no be1.domBE & be2.domBE
}

The Alloy code used to specify transactions is the following:

sig Transaction {
type: TransactionType,
bE: one BusinessEvents,
eventOrder: EventOrder,
} {
```
--the set of events associated to each transaction has to
--contain events of type Initiation and Termination compliant
--with its type
```
this.eventSet.type & InitiationEventType in type.boundaries.iEventTypes
this.eventSet.type & TerminationEventType in type.boundaries.tEventTypes
}
```
--all events of type ObjectEvent belonging to a transaction
--domain, also belong to the domain of the BusinessEvents
--associated to the transaction
```
fact {all e:Event|some t:Transaction|((e.type in ObjectEventType) and (e in t.eventSet)) =>(one be:BusinessEvents |t.bE=be and e in t.bE.domBE)}
fact transactionsHaveDisjointEventSets {

```
all disj t1, t2: Transaction | no t1.eventSet & t2.eventSet
}
fact transactionsHaveDisjointBEs {
all disj t1, t2: Transaction| no t1.bE & t2.bE
}
 sig EventOrder { -- strict partial order on events
        domain: set Event,
        r: Event -> Event
}{
r.Event + Event.r in domain
all e: domain| e->e not in r -- irreflexivity
all  e1,e2: domain| e1->e2 in r => not e2->e1 in r -- asymmetry
-- transitivity
all e1,e2,e3: domain| (e1->e2 in r and e2->e3 in r) => e1->e3 in r
--an atom of this signature is present in the instances only in case
-- it  is associated to a transaction or to the History
(some t: Transaction|  t.eventOrder= this) or History=this
}
pred EventOrder::eventSource[e1: Event] {
        no e: Event| e->e1 in this.r
}--an event is a source if no event preceding it exists
pred EventOrder::eventSink[e1: Event] {
        no e: Event| e1->e in this.r
}--an event is a sink if no event following it exists
pred EventOrder::precedes[e1,e2: Event] { -- e1<e2
        e1->e2 in this.r
}
```

The Alloy code used to specify a computation of a set of (concurrent) transactions is the following:

```
one  sig History extends EventOrder {
-- partial order over set of events of a set of transactions
} {
domain = { e: Event| some t: Transaction| e in t.eventSet }
all t: Transaction| all e1, e2: t.eventSet | t.eventOrder.precedes[e1,e2] iff this.precedes[e1,e2]

all disj e1,e2: domain | e1.type=ObjectEventType and e2.type=ObjectEventType and e1.op.ob = e2.op.ob=> ( precedes[e1,e2] or precedes[e2,e1])
}
```

Sat analysis: histories compliant with the Flat model

The predicate showFlatHistories used to direct the Alloy Analyzer is the one shown in the following:

```
pred showFlatHistories {
some disj t1, t2: Transaction | some disj e1, e2, e3, e4, e5, e6: Event | t1.type=Flat and t2.type=Flat
and t1.type+t2.type in FlatModel.t
}
```

This predicate, informally, requires the Analyzer to find instances in which there are two disjoint transactions (compliant with the flat model) and six disjoint events.

The instances found by the tool, like the one presented in Figure 3, correspond to the set of allowed computations that could be obtained manually by considering all the possible orderings compliant with the simplified flat model. The analysis is executed by limiting the scope to 6 atoms.

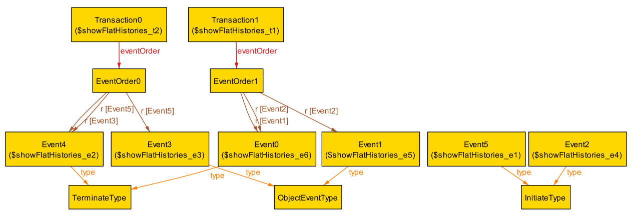

Transaction0 ($showFlatHistories_t2)
Transaction1 ($showFlatHistories_t1)
eventOrder
eventOrder
EventOrder0
EventOrder1
r [Event5]
r [Event3]
r [Event5]
r [Event2]
r [Event1]
r [Event2]
Event4 ($showFlatHistories_e2)
Event3 ($showFlatHistories_e3)
Event0 ($showFlatHistories_e6)
Event1 ($showFlatHistories_e5)
Event5 ($showFlatHistories_e1)
Event2 ($showFlatHistories_e4)
type
type
type
type
type
type
TerminateType
ObjectEventType
InitiateType

Figure 5 Instance of a flat transactional computation

From Figure 5, it can be seen that the r relation orders events as follows:

Transaction0: Event5<Event3<Event4
Transaction1: Event2<Event1<Event0

The boundary's constraints are, therefore, satisfied. For each transaction, in fact, there is only one event of type InitiateType (namely, Event5 for Transaction0 and Event2 for Transaction1) and this event represents the source in the ordering relation r. For each transaction, there is only one event of type TerminateType (namely, Event4 for Transaction0 and Event0 for Transaction1) and this event represents the sink in the ordering relation r. The remaining two events (Event3 and Event1) are of type Object Event Type and they are preceded by the events marking the initiation and followed by the events marking the termination.

Sat analysis: histories compliant with the Nested model

The predicate showNestedHistories used to direct the Alloy Analyzer is the one shown in the following:

```
pred showNestedHistories {
some disj t1, t2:Transaction|
t1.type =Root &&t2.type = Child and t1.type+t2.type in NestedModel.t&&(all e:t2.eventSet|e.source=t1)
}
```

This predicate, informally, requires the Analyzer to find instances in which there are two disjoint transactions, *t1* and *t2*, compliant with the simplified Nested transactions model. Moreover, *t1* has to be of type *Root* and *t2* has to be of type *Child* and it has to be invoked by *t1*.

The instances found by the tool, like the one presented in Figure 6, correspond to the set of allowed computations that could be obtained manually by considering all the possible orderings compliant with the nested model. The analysis is carried out by limiting the scope to 6 atoms.

From Figure 6, it can be seen that, to satisfy the boundary's constraints, four events are events marking the boundary (Event0 and Event1 mark the boundary of Transaction1 of type *Root*; while Event3 and Event2 mark the boundary of Transaction0 of type *Child*). The remaining two events (Event4 and Event5) are of type Object Event Type.

Each transaction satisfies the standard boundary axioms. In fact we can see that the events belonging to the transaction domain are ordered (through the r relation) correctly.

Transaction0: Event3<Event5<Event2
Transaction1: Event0<Event4<Event1

The nesting between the two transactions appears in the ordering relation r of the History. In fact we can see that Event0 precedes Event3 and that Event2 precedes Event1.

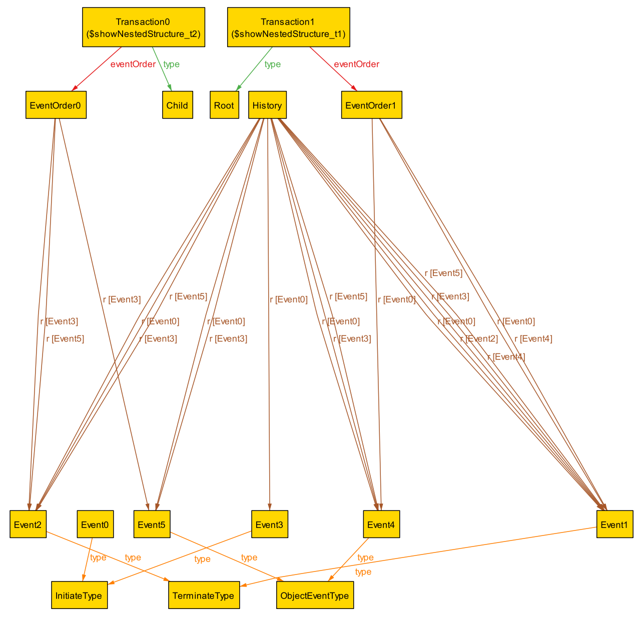


Figure 6 Instance of a nested transactional computation

Sat analysis: serializable histories

The predicate showFlatSerializableHistories used to direct the Alloy Analyzer is the one shown in the following:

pred showFlatSerializableHistories {

some disj t1, t2:Transaction| some disj e1, e2, e3, e4, e5, e6, e7, e8: Event | one oB:Object|

e2 in t1.eventSet && e3 in t1.eventSet && e4 in t2.eventSet && e5 in t2.eventSet && e2.op.type=Write && e3.op.type=Write && e4.op.type=Write && e5.op.type=Write && (e2.op.ob)=oB && e3.op.ob=oB && e4.op.ob=oB && e5.op.ob=oB

}

This predicate, informally, requires the Analyzer to find instances in which there are two disjoint Flat transactions and eight disjoint events. Four events have to execute an operation of type write on the same object. Among these four events, two have to belong to one transaction and two to the other.

Limiting the analysis, by considering only two transactions, is coherent with the small-scope hypothesis [13]. Two transactions are enough to reveal problems related to interference. The instances found by the tool correspond to the set of allowed computations that could be obtained manually by considering serializable computations in the classical formal framework introduced to reason about serializability theory. Because of space reasons we do not present any instances. The interested reader, however, could execute the complete model available in the technical report related to this paper [17].

6. Related works

Since the early nineties of the past century, researchers have recognized that, to increase reuse of transactional core features, it is crucial not only to reason at the meta-model level in order to identify similarities and differences among TMs, but also to do it formally.

The ACTA framework was the first effort in that direction. ACTA is a semi-formal framework for the specification and synthesis of TMs on the basis of building blocks. As discussed in [8], however, it presents several weaknesses. Other works, mainly ACTA descendants, have been proposed to address some of the lacks that ACTA itself presented and some of them have already been discussed in the related works presented in [8].

A factorization of transactions into individual features (lock-based serializability, undoability, persistence, etc.) is provided in [12, 16]. This work provides some high level transactional constructs which extend the programming language StandardML. Even though this work contributes in highlighting the potential of achieving a general-purpose control abstraction, with which variations of the transactional model could be built, no further development has followed.

A generalization of the notion of transaction, called generalized advanced transaction, is provided in [15]. A generalized advanced transaction is defined as a directed graph having as nodes a set of transactions and as edges a set of dependencies. This work focuses on a detailed characterization of the dependencies (structural and behavioural) that may relate two transactions and on their correct scheduling. With respect to the ACTA framework, this work better investigates the notion of behavioural dependency and introduces further useful notions for the analysis of dependencies (i.e. conflict between two dependencies; inclusion of one dependency in another; redundant dependency). This work, however, does not address the meta-model level.

An equational theory for transactions is provided in [3]. In this work authors define three categories of actions (A-actions, I-actions and D-actions) which respectively capture the Atomicity, Isolation and Durability properties. A further category of actions to capture the Consistency property is not introduced. Consistency is expressed as an induction rule and can be derived. The three kinds of actions can be nested leading to different TMs. The categories classifying the actions however are not further decomposed and therefore in this approach it is not possible to obtain several variants. A property (i.e. atomicity) either characterizes an action or not. The TMs which can be defined according to this theory are only those which have a set of actions belonging to the power set of the set containing the three categories as elements (A, I, and D Actions).

A specification of TMs at the logical level by appealing to non Markovian theories of the situation calculus is provided in [14]. In this work authors not only provide a semantics for the ACTA building blocks using the situation calculus but they also specify further important aspects such as integrity constraints. Finally they simulate their specifications using GOLOG language support environment. This work also improves the ACTA framework towards a tool support for specification simulation; however, its focus is not centred in organizing building blocks around ACID properties and within an SPL perspective.

7. Conclusion and future work

In this paper, we have provided a first step towards the provision of a formal semantics for the SPLACID language by translation into Alloy. We have presented general rules to translate the SPLACID concepts into Alloy concepts and then we have applied them focusing on those concepts pertaining to the structure of a TM and those characterizing the isolation variant. The Alloy specifications obtained by this translation preserve the SPLACID main key-properties consisting of modularity, flexibility and reusability. To support this claim we have shown how flexible structures and flexible isolation variants can be obtained in Alloy thanks to reusable and modular pieces of specification. Finally we have discussed the results achieved by carrying on the satisfiability analysis of the Alloy model. The analysis in this first step has been limited to two aspects: 1) the skeleton of the Flat and Nested transactions model to focus on constraints related to the boundary and 2) the serializability variant in the Flat transactions model.
In the immediate future we aim at further improving the engineering of the SPLACID language. In particular, we will provide an abstract syntax in terms of a meta-model and we will strengthen the integration of the SPL perspective within the Alloy formalization by making explicit the variability modelling, as proposed, for instance, in [2].
We also aim at formalizing in Alloy various isolation variants [1] to be used in a structure-less as well as in a structured TM (i.e. Nested transactions). The formalization of the other variation points (atomicity, consistency and durability) as well as the associated variants is also an immediate future goal. Our ultimate and long-term goal is to be able to verify the soundness of a TM (product of the TMsPL) derived by selection and composition of available pieces of specification corresponding to the transactional features and to assign an ACIDity measure to it.

References

[1] A. Adya, B. Liskov, and P. ÓNeil. Generalized isolation level definitions. In 16th IEEE International Conference on Data Engineering (ICDE' 00), pages 67–80, Washington - Brussels - Tokyo, March 2000.

[2] R. Gheyi and T. Massoni and P. Borba. A Theory for Feature Models in Alloy. First Alloy Workshop, Portland, Oregon, November 6, 2006.

[3] Black, Cremet, Guerraoui, and Odersky. An equational theory for transactions. Foundations of Software Technology and Theoretical Computer Science, 23, 2003.

[4] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. ACM Trans. Database Syst., 19(3):450–491, 1994.

[5] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In Robert L. Nord, editor, Proceedings of the Third Software Product Line Conference, LNCS 3154, pages 266–283, Boston, MA, September 2004.

[6] A. K. Elmagarmid, editor. Database Transaction Models for Advanced Applications. Morgan Kaufmann, San Mateo , CA , USA, 1992.

[7] B. Gallina and N. Guelfi. A product line perspective for quality reuse of development frameworks for distributed transactional applications. In COMPSAC, pages 739–744. IEEE Computer Society, 2008.

[8] B. Gallina and N. Guelfi. Splacid: an SPL-oriented, ACTA-based Language, for Reusing (Varying) ACID Properties. In The 32nd IEEE International Workshop on Software Engineering, Porto Sani, Greece, ISBN 978-0-7695-3617-0, pages 115-124, 2008.

[9] J. Gray. The transaction concept: Virtues and limitations. In Proc. Int'l. Conf. on Very Large Data Bases, page 144, Cannes, France, September 1981.

[10] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In Working IEEE/IFIP Conference on Software Architecture (WICSA), Washington, DC, USA, 2001.

[11] V. Hadzilacos. A theory of reliability in database systems. JACM: Journal of the ACM, 35, 1988.

[12] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. ACM Transactions on Programming Languages and Systems, 16(6):1719–1736, November 1994.

[13] D. Jackson. Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge, Mass., 2006.

[14] I. Kiringa. Simulation of advanced transaction models using GOLOG. LNCS, 2397:318–341, 2002.

[15] I. Ray and T. Xin. Analysis of dependencies in advanced transaction models. Distributed and Parallel Databases, 20(1):5–27, July 2006.

[16] J. Wing. Decomposing and recomposing transaction concepts. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, Object-Based Distributed Programming ECOOP '93 Workshop, Kaiserslautern, Germany, July 26-27, 1993, volume 791 of LNCS, pages 111–122. Springer, Berlin, 1994.

[17] B. Gallina, N. Guelfi, Towards an Alloy Formal Model for Flexible and Reusable Advanced Transactional Model Development, Technical Report, LASSY, University of Luxembourg, (to be sub-mitted).